October 1983

Three Dimensional Graphics Application of the iAPX 86/20 Numeric Data Processor

Ken Shoemaker
Microcomputer Applications

Order Number: 210384-001

INTRODUCTION

As the performance of microcomputers has improved, these machines have been used in many applications. With the introduction of 16-bit microprocessors (along with the associated CPU enhancements, especially the integer multiply instruction) the operations required to manipulate graphic representations of three-dimensional objects were made easier. Only integer values could be used to define figures, however, because only integer multiplies were supported in hardware. While software floating point routines existed, the speed at which a general purpose microprocessor could execute even the simplest floating point operation precluded the use of these routines because of the number of floating point operations which must be performed to manipulate all but the simplest of objects.

The lack of high performance floating point math or the restriction of using only integer representations severely limits the types and sizes of objects that can be defined. Imagine limiting everything in the universe to be less than 32,000 millimeters long, high, or wide! This limitation could severely impact any system that is used to model real world objects. An example of such an application is a Computer Aided Design (CAD) system. If real or floating point numbers are used, however, practically any object can be defined (after all, there are only 9,397,728,000,000,000,000 millimeters in a light year(!), well within the range of floating point numbers). With the introduction of the iAPX 86/20, the performance required to execute the requisite operations on floating point representations of three-dimensional figures has finally been achieved in a microprocessor solution, at a microprocessor price.

The iAPX 86/20 features the Intel 8086 with the 8087 numerics co-processor. This combination allows for high performance, high precision numeric operations. This performance is especially important in the graphics routines implemented in this note because of the large number of floating point operations performed for each line drawn. In addition, the precision is required to maintain the image quality of the represented figures.

This application note shows the fundamental components of a three-dimensional graphics package. As stated before, if the objects are to be described in real size, floating point values must be used. Since the operations performed require many multiplies and divides, a high performance floating point arithmetic unit is a must. Note that the operations to be performed by this software are not those of a "bit map" controller: single chip devices performing this specialized task are or will soon be available. Because they are special-purpose devices, they can also execute this task quickly, offloading the task from the general purpose microprocessor allowing the processor to perform other work in parallel. In addition, since the size of the memory used in a bit-mapped controller is constrained (one could hardly have unlimited memory for the refresh map), only integer math is required. This graphics package is a much higher level type of routine, where the inputs are three-dimensional line drawing commands (which could be fed into a bit map controller).

The three-dimensional graphics package implemented in this note allows for the entry of three-dimensional figures, the manipulation of these figures, the setting of the viewer's location, the size of the picture to be seen, and the position of the picture on the graphics output device. Along the way, it performs perspective transformations, window clipping and projection. All figures are defined using floating point numbers. Thus, any figure may be defined "real size" without pre-scaling. This means that the size of the figure defined within the package may be the actual size of the object, i.e. the size of the object is not arbitrarily limited by the machine, whether the object be a sub-nuclear particle, or a cellestial body.

iAPX 86/20 HARDWARE OVERVIEW

The iAPX 86/20 is a 16-bit microprocessor based on the Intel 8086 CPU. The 8086 CPU features eight internal general purpose 16-bit registers, memory segmentation, and many other features allowing for compact, efficient code generation from high-level language compilers. When augmented with the 8087, it becomes a vehicle for high-speed numerics processing. The 8087 adds eight 80-bit internal floating point registers, and a floating point arithmetic logic unit (ALU) which can speed floating point operations by up to 100 times over other software floating point simulators or emulators.

The 8086 and 8087 execute a single instruction stream. The 8087 monitors this stream for numeric instructions. When a numeric instruction is decoded, the 8086 generates any needed memory addresses for the 8087. The 8087 then begins instruction execution automatically. No other software interface is required, unlike other floating point processors currently available where, for example, the main processor must explicitly write the floating point numbers and commands into the floating point unit. The 8086 then continues to execute non-numeric instructions until another 8087 instruction is encountered, whereupon it must wait for the 8087 to complete the previous numeric instruction. The parallel 8086 and 8087 processing is known as concurrency. Under ideal conditions, it effectively doubles the throughput of the processor. However, even when a steady stream of numeric insructions is being executed (meaning there is no concurrency), the numeric perfor-

mance of the 8087 ALU is much greater than that of the 8086 alone.

The hardware interface between the 8086 and the 8087 is equally simple. Hardware handshaking is performed through two sets of pins. The RQ/GT pin is used when the 8087 needs to transfer operands, status, or control information to or from memory. Because the 8087 can access memory independently of the 8086, it must be able to become the "bus master," that is, the processor with read and write control of all the address, data and status lines.

The TEST/BUSY pin is used to manage the concurrency mentioned above. Whenever the 8087 is executing an instruction, it sets the BUSY pin high. A single 8086 instruction (the WAIT instruction) tests the state of this pin. If this pin is high, the WAIT instruction will cause the 8086 to wait until the pin is returned low. Therefore, to insure that the 8086 does not attempt to fetch a numeric instruction while the 8087 is still working on a previous numeric instruction, the WAIT instruction needs to precede most numeric instructions (the only class of instructions which do not need to be preceded by a WAIT instruction are those which access the control registers of the 8087). The 8086/87/88 assembler, in addition to all INTEL compilers, automatically inserts this WAIT instruction before most numeric instructions. Software polling can be used to determine the state of the BUSY pin if the hardware handshaking is not desired.

Most other lines (address, status, etc.) are connected directly in parallel between the 8086 and the 8087. An exception to this is the 8087 interrupt pin. This signal must be routed to an external interrupt controller. An example iAPX 86/20 system is shown in Figure 1. A more complete discussion of both the handshaking protocol between the 8086 and the 8087 and the internal operation of the 8087 can be found in the application note *Getting Started With the Numeric Data Processor,* Ap Note #113 by Bill Rash, or by consulting the numerics section of the July 1981 *iAPX 86, 88 Users Manual.*

In addition to the 8087 hardware, the 8086 is also supported by Intel compilers for both Pascal and FORTRAN. Code generated by these compilers can easily be combined with code generated from the other compiler, from the Intel 8086/87/88 macro assembler, or the Intel PL/M compiler. In addition, these compilers produce in-line code for the 8087 when numeric operations are required. By producing in-line code rather than calls to floating point routines, the software overhead of an unnecessary procedure call and return is eliminated.

The combination of both hardware co-processors and software support for the iAPX 86/20 provides for greater performance of the end product, and a quicker, easier development effort.

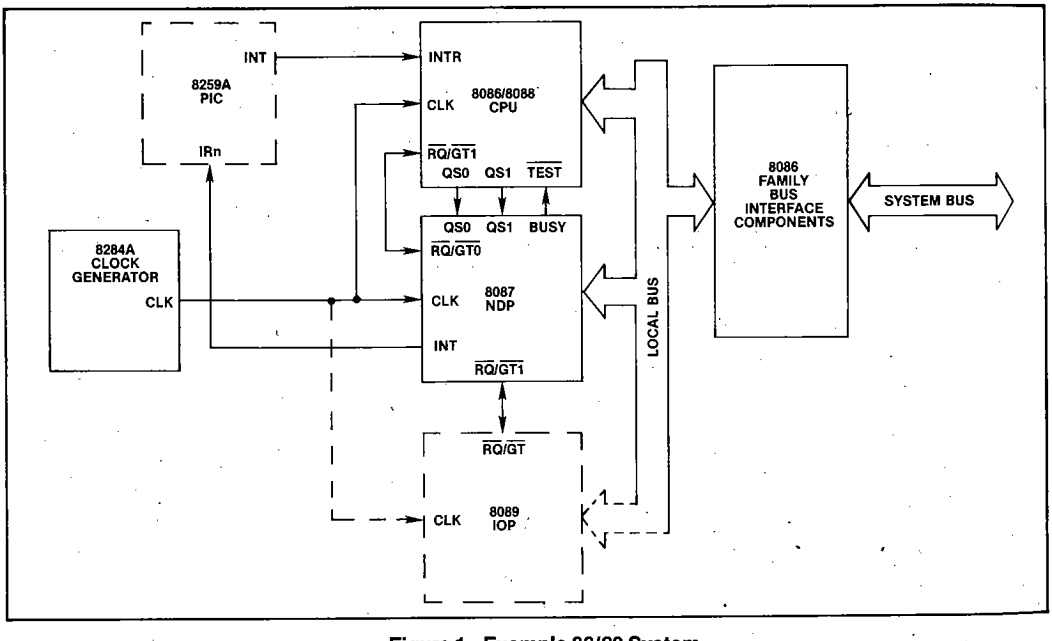

Figure 1. Example 86/20 System

THREE-DIMENSIONAL GRAPHICS FUNDAMENTALS

The charter in life of a three-dimensional graphics package is to take a three-dimensional rendering of an object and to transform it such that it can be accurately represented on the two-dimensional surface of a graphics output device. To fulfill these requirements, the graphics package must:

- **Allow for the entry of three-dimensional data.** Since all figures inside the package are represented as a series of points in three-dimensional space, there must be a way of entering these figures into the computer.
- **Perform the current transformation.** This transformation rotates, translates and scales the three-dimensional object throughout three-dimensional space. Example rotates, translates and scales are shown in Figures 2–11. In all diagrams, the first coordinate indicated is X, the second Y, the third Z. The viewpoint is the location of the viewer in three-dimensional space in relationship to an arbitrarily chosen but consistent origin.

  Translations are movements of the object in three-dimensional space. Example translations are shown in Figures 3–5. Figure 3 shows a translation of two units in the plus Z direction. Since the viewpoint is ten units up along the Z axis, this moves the cube one-fifth the distance toward the viewer, or in other words, the cube seems to get larger. Figure 4 shows the same cube translated two units in the plus X direction. Since the cube is four units on a side, this moves the cube such that the viewer is looking straight down one side of the cube. The viewer is also looking straight down a side in Figure 5.

  Rotations are movements of the object in three-dimensional space about the three-coordinate axis: X, Y, and Z. The rotation of the object must specify both the magnitude of the rotation, and the axis about which the rotation must take place. Example rotates are shown in Figures 6–8. Figure 6 shows the cube rotated 45 degrees about the Z axis. Since the viewpoint is straight up the Z axis, the cube is seen to keep its same face towards the viewer. Figure 7 shows the cube rotated 45 degrees about the X axis. Here, the cube no longer shows the same face it has previously. The face previously turned directly toward the viewer has been rotated such that the edge between this face and another face is immediately before the viewer. The same is also shown in the rotation about the Y axis in Figure 8.

  Scaling is the multiplication of all coordinates of the points defining a figure by a constant number such that the object becomes larger or smaller. Example scales are shown in Figures 9–11. This scaling need not be uniformly performed for all dimensions of an object. If, for example, the Z coordinates of a cube are all scaled to be twice as large as they originally were, the image shown in Figure 9 would be produced. Notice here that the X and Y coordinates have not been altered; only the Z coordinates are twice as large as they originally were, or alternatively, the front and back of the cube are closer and farther away from the viewer than in the original, unaltered cube. Figure 10 shows this same operation being performed on the X coordinates, while Figure 11 shows this operation being performed on Y coordinates.

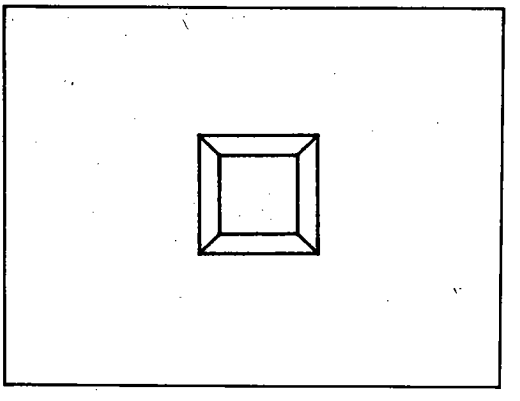

Figure 2. 2 x 2 x 2 Cube Centered at (0,0,0) Viewed from (0,0,10)

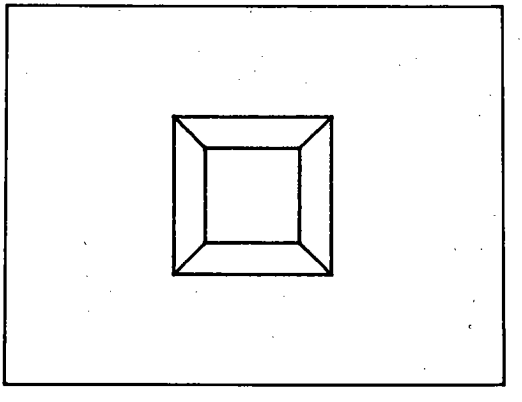

Figure 3. Same Cube and Viewpoint, +2 Z Translation

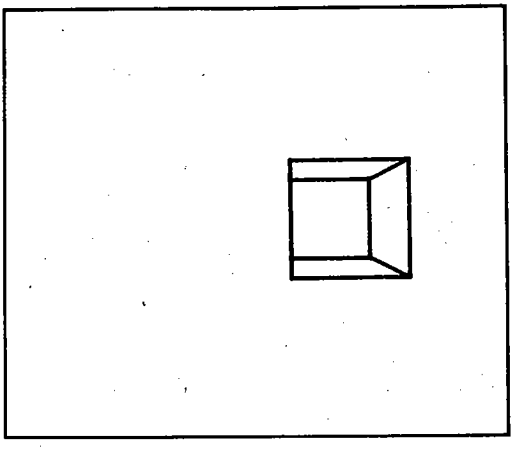

Figure 4. Same Cube, Viewpoint, +2 X Translate

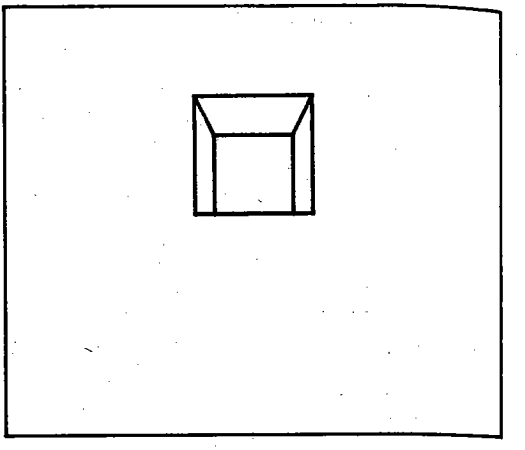

Figure 5. Same Cube, Viewpoint, +2 Y Translate

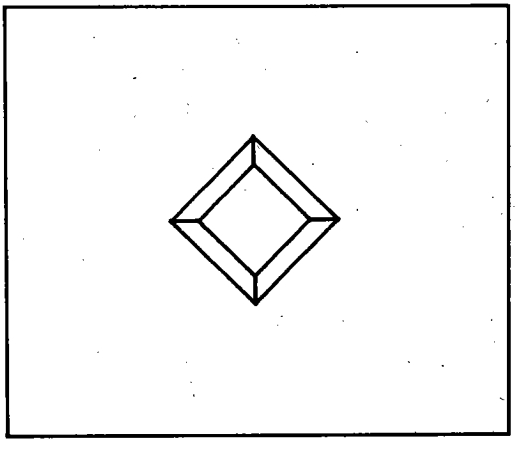

Figure 6. Same Cube, Viewpoint, 45 Degree Rotation About Z

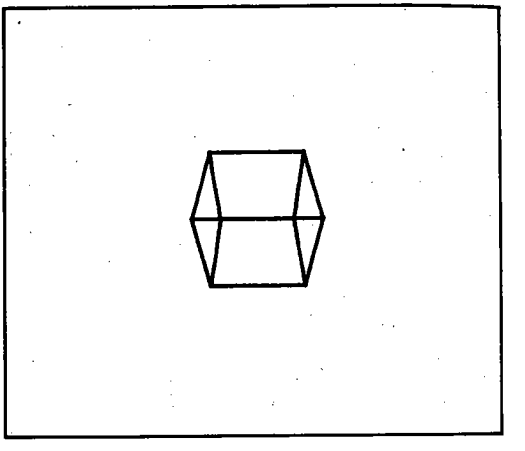

Figure 7. Same Cube, Viewpoint, 45 Degree Rotation About X

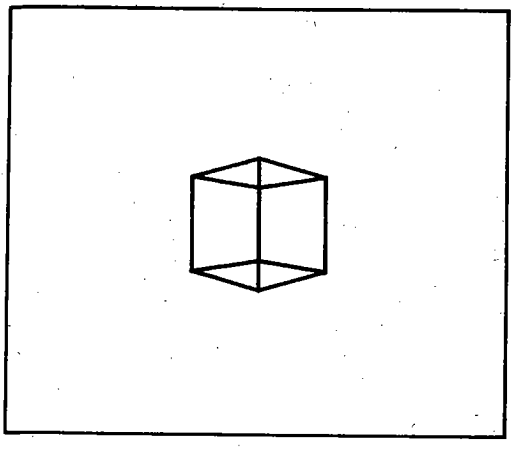

Figure 8. Same Cube, Viewpoint, 45 degree Rotation About Y

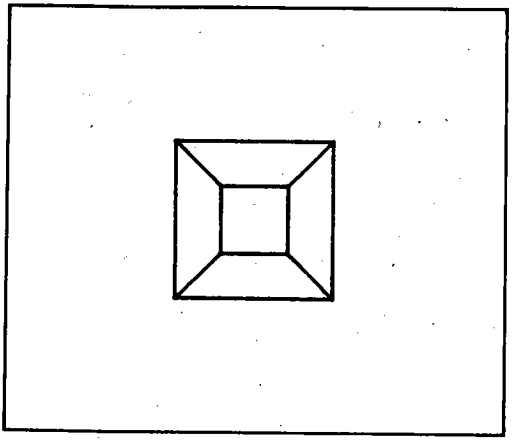

Figure 9. Same Cube, Viewpont 2 x Scale of Z

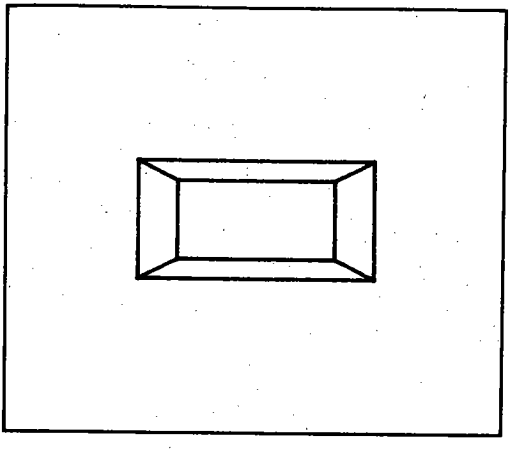

Figure 10. Same Cube, Viewpoint, 2 x Scale of X

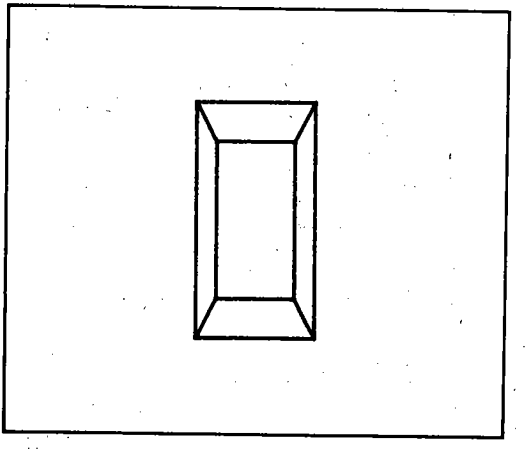

Figure 11. Same Cube, Viewpoint, 2 x Scale of Y

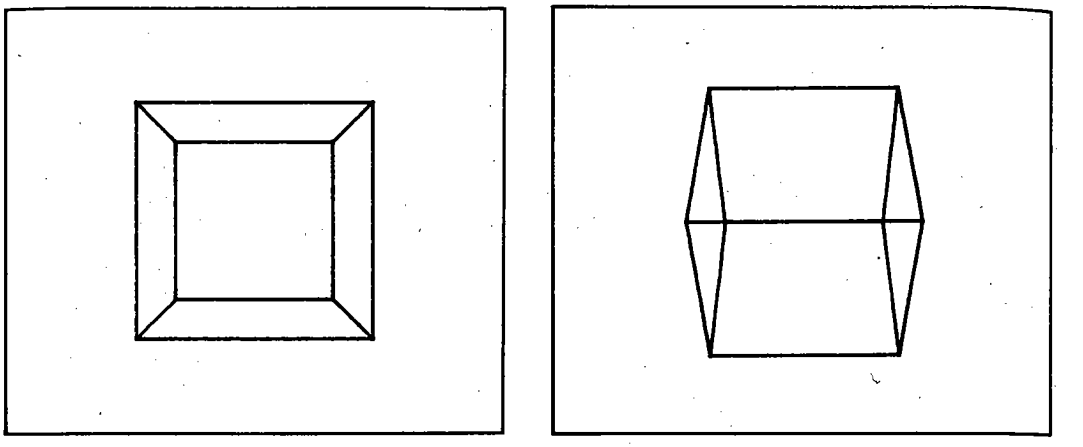

Figure 12. 2 x 2 x 2 Cube Centered at (0,0,0) Viewed from (0,0,10) Then from (10,10,0)

- **Perform the viewing transformation.** This transformation moves and rotates the three dimensional figure according to the viewer's location and orientation (the direction the viewer is facing) in space. An example of changing the view location is shown in Figure 12. Again, this location, or viewpoint, is the viewer's location with relation to an arbitrarily chosen origin.
- **Perform Z-clipping on the three-dimensional data.** This insures that only data in front of the viewer are displayed. In addition, it allows that objects beyond a certain distance from the viewer will not be displayed.
- **Project the three-dimensional data onto a two dimensional surface.** The objects must be projected onto a two-dimensional surface according to the laws of perspective. By changing the "vanishing point," interesting effects are also possible. An example of this is shown in Figure 13. Here, the first figure shows exaggerated perspective (that is, the difference in perceived size between the front face and the back face of the cube is exaggerated), where the second figure shows the object with subdued perspective (the difference in the perceived sizes of the front and back faces is much less than in the first figure). Exaggerated perspective is generated for objects close to the viewer, while subdued perspective is generated for objects distant from the viewer. Note that the same figure, with the same dimensions, is shown in both figures; only the perspective values have been changed.
- **Perform X-Y clipping on the projected data.** This cuts off lines in the projected data extending beyond the specified "window."
- **Perform the window to viewport transformation.** This takes the two-dimensional projected values and scales them according to the relative sizes of the "window" and the "viewport."

The "window" describes the size of the viewer's portal into the data, whereas the "viewport" describes the size and position of this portal on the graphics output device. Whereas the window's size is determined by the size of the input data, the viewport size is determined by the physical characteristics of the graphics display device. For example, the viewport coordinates of a certain CRT display may be constrained to be between 0 and 1023 in both the X and Y dimensions, whereas the window limits are determined only by the maximum size of numbers the computer can store. Thus, for maximum generality and utility, floating point numbers must be used to represent the three-dimensional figures.

A good reference to the techniques used in this three-dimensional graphics implementation can be found in Newman and Sproull[1].

[1] Newman, William M. and Robert F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill Book Company, New York, 1979.

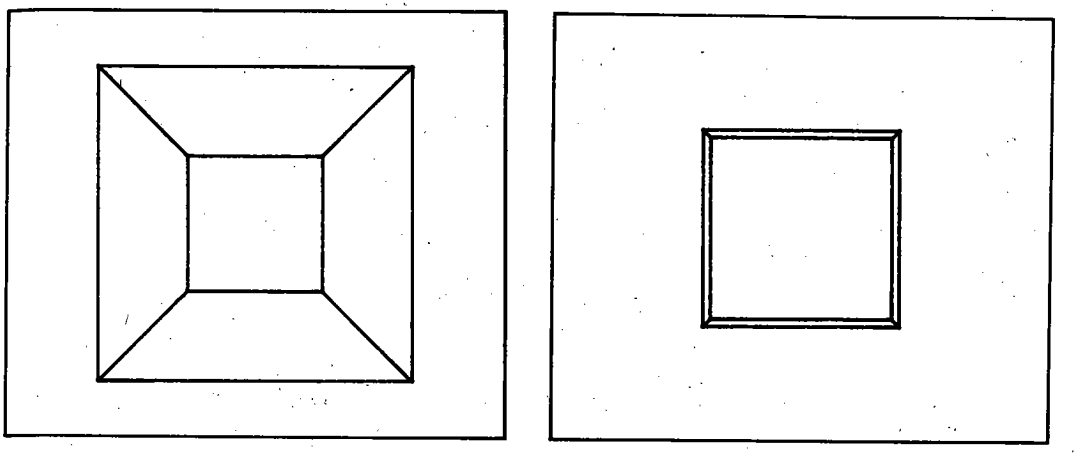

Figure 13. Example Cube Shown with Exaggerated Perspective, then with Subdued Perspective

IMPLEMENTATION

Three-dimensional graphics systems can be split into three functional modules: the input hardware, the processing hardware, and the output hardware. The graphics software is executed by the processing hardware and is used to receive figure definitions from the input hardware, store them in one form or another, and manipulate them such that they can be displayed on the output hardware.

Input hardware can range from the common typewriter keyboard to sophisticated three-dimensional input devices. Output hardware can range from a plotter to a storage tube terminal to a bit-mapped raster scan display or a vector drawing CRT.

The processing hardware can range from general purpose minicomputers to very fast, specialized graphics processing hardware. General purpose computers are used because they allow applications programs to be written in higher level languages. Specialized hardware is sometimes employed when very fast manipulations are required, such as in the real time graphics applications found in flight simulators. This specialized hardware can be used to perform whole matrix transformations. Many applications do not require figures to be drawn real time (on the order of one complete picture every 1/30 sec), however, and can be satisfied by the performance of the general purpose computer alone. A typical application which is satisfied by these latter requirements is a Computer Aided Design (CAD) system. However, since these graphics systems often exist in an interactive environment, picture processing delays greater than a few seconds for simple figures, or greater than a few minutes for very complex figures cannot be tolerated. Because of these processing requirements, a mini-computer with a hardware floating point unit has been required to drive these graphics systems. However, with the introduction of the 8087, the floating point processing performance required by these systems can finally be met in a microcomputer solution.

The microcomputer system used in this three-dimensional graphics application is a general purpose microcomputer embodied in the iAPX 86/12 board found in an Intel Intellec® Series III development system. All routines implemented in this application note were written entirely in FORTRAN using the Intel FORTRAN 86 compiler. Any iAPX 86/20 (or iAPX 88/20) with enough memory can be used to execute the programs, however. The amount of memory required depends on the number and complexity of the figures to be displayed. The source code for all routines used in this note are given in the appendix.

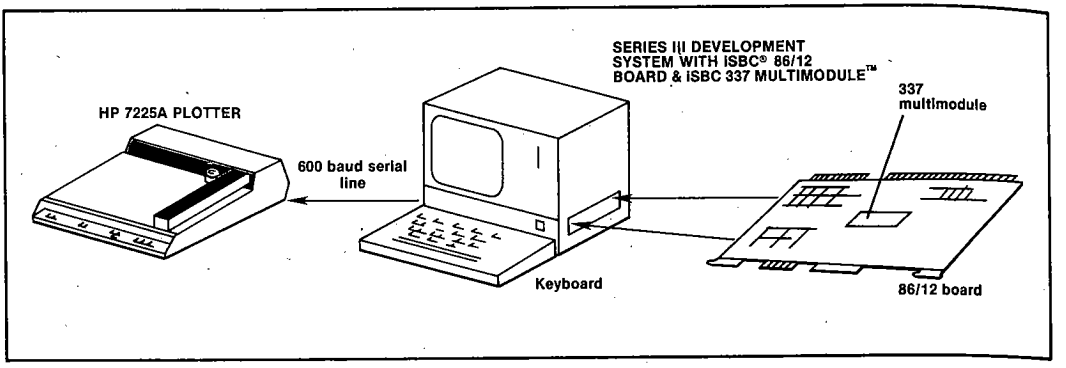

Figure 14. Computer System Used in This Graphics Implementation

The graphics output device used was a HP 7225A flat bed plotter. Communications were performed using the RS232 serial link on the 86/12 board. The communications speed of the line to the plotter was 600 baud. Because of the number of lines drawn in the more complex figures, the physical characteristics of the plotter, and the communications line speed, the amount of time required to draw a large picture was a function of the plotter speed, not the execution speed of the iAPX 86/20. As a result, all times quoted in this note do not reflect the plotting time. Only the time up to placing the ASCII character into the buffer of a serial communications chip is included for all machines quoted. Higher speed graphics display devices (which are not limited by the physical characteristics of plotters) can use the speed of the iAPX 86/20 to full advantage.

The graphics input device used was the standard alphanumeric keyboard attached to the development system. This allows entry of figures, as well as control of the graphics system. Input can also be fetched from disk storage, however, to allow for greater speed in defining large figures. A block diagram of the hardware system used in this implementation is shown in Figure 14.

All routines were run using both the 8087 and the 8087 software emulator. The 8087 software emulator is a software package exactly emulating the internal operation of the 8087 using 8086 instructions. When the emulator is used, an 8087 is not required. The emulator is a software product available from Intel as part of the 8087 support library. The performance of the 8087 hardware is much better than that of the software emulator, as one would expect from a specialized hardware floating point unit.

The 8087 supports various data formats. For real numbers, these formats are short real (or single precision), long real (or double precision), and temporary real (or extended precision). The differences among the three are in the number of bits allocated to represent a given floating point number.

In all real numbers, the data is split into three fields: the sign bit, the exponent field and the mantissa field. The sign bit shows whether the number is positive or negative. The exponent and mantissa together provide the value of the number: the exponent providing the power of two of the number, and the mantissa providing the "normalized" value of the number.

A "normalized" number is one that always lies within a certain range. By dividing a number by a certain power of two, most numbers can be made to lie between the numbers 1 and 2. The power of two by which the number must be divided to fit within this range is the exponent of the number, and the result of this division is the mantissa. This type of operation will not work on all numbers (for example, no matter what one divides zero by, the result is always zero), so the number system must allow for these certain "special cases."

As the size of the exponent grows, the range of numbers representable also grows, that is, larger and smaller numbers may be represented. As the size of the mantissa grows, the resolution of the points within this range grows. This means the distance between any two adjacent numbers decreases, or, to put it another way, finer detail may be represented. Short real numbers provide 8 exponent bits and 23 significand or mantissa bits. Long real numbers provide 11 exponent bits and 52 significand bits. Temporary real numbers provide 15 exponent bits and 64 significand bits. These data formats are shown in Figure 15. Thus, of the three data formats implemented, short real provides the least amount of precision, while temporary real provides the greatest amount of precision. These levels of precision represent only the external mode of storage for the numbers; inside the 8087 all numbers are represented to temporary real precision. Numbers are automatically converted into the temporary real precision when they are loaded in-

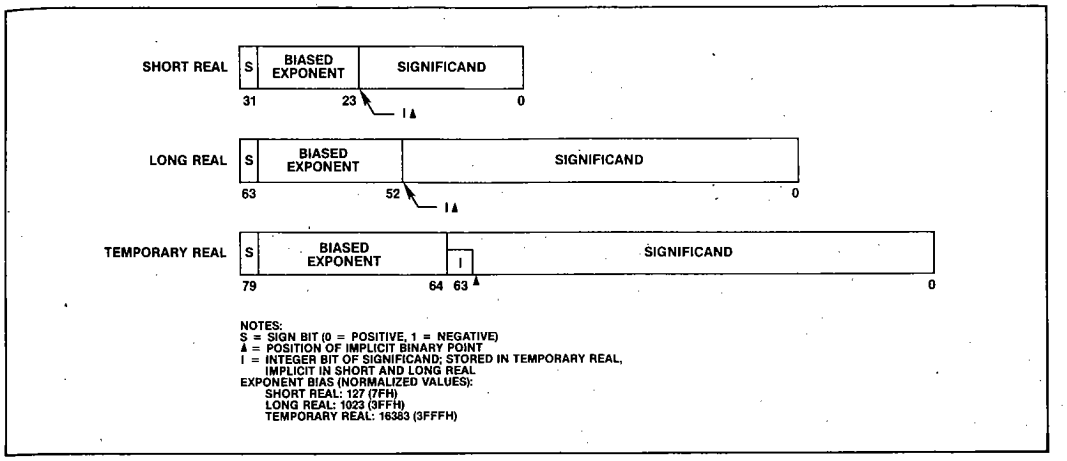

Figure 15. Floating Point Data Formats

to the 8087. In addition to real format numbers, the 8087 automatically converts to and from external variables stored as 16, 32 or 64-bit integers, or 80-bit binary coded decimal (BCD) numbers.

Memory requirements also increase as precision increases. Whereas a short real number requires only four bytes of storage (32 bits), a long real number requires eight bytes (64 bits) and a temporary real number ten bytes (80 bits). In many floating point processors, processing time also increases dramatically as precision is increased, making this another consideration in the choice of precision to be used by a routine. The differences in 8087 processing time among short real, long real and temporary real numbers are insignificant compared to the processing time, however, since all operations are performed to the internal 80-bit precision. This makes the choice of which precision to use in an iAPX 86/20 system a function only of memory limitations and precision requirements.

Double precision numbers were chosen for this graphics implementation because they allow a very wide range of numbers to be represented with high precision. This is important, since the package allows the user to magnify small parts of defined figures. Without the precision gained by using double precision numbers, the image of the object could easily be distorted under such scrutiny.

Three-Dimensional Figure Description and User Interface

The graphics user interface implemented in this note is both functional and simple. It does not require the use of specialized three-dimensional input hardware. All input data is keyed in through the keyboard.

The package allows for definition of figures for future use within the graphics package. This feature could be useful in generating multiple views of a certain object. It requires that the object be "defined" at the beginning of the session, but then allows the user to view the object from any location, with any rotation, scale, or translation.

Commands to the graphics package consist of a set of alphanumeric commands followed by the necessary numeric constants. To enter commands to the graphics package, one enters an alphanumeric command enclosed within the single quotes followed by the appropriate numeric arguments. The maximum number of arguments required by any command is six. If less than six arguments are entered on a line, the line must be terminated by the '/' character, however. These requirements (having the command enclosed within single quotes, explicitly terminating the line) are a result of using the list-directed input format of FORTRAN.

The commands recognized by the graphics processor are:

comment *arg1.* This command instructs the graphics processor to ignore the next *arg1* lines. This can be used to insert comments within the graphics commands.

define *arg1.* This command instructs the graphics processor that the next N lines (up to the **enddef** command) are to be entered into an internal buffer for future reference as figure *arg1.* The graphics commands are not interpreted, i.e. they do not cause figures to be drawn as they are entered. In this way, three-dimensional objects may be defined, or to put it another way, placed into an internal display list. Up to ten objects may be defined using the current version of the program. This may be increased to the limits of available memory. Currently there is internal storage space for up to 500 total graphics commands. These may be spread in any combination among the ten figures. This number may also be modified to reflect memory restrictions.

enddef. This command terminates a figure definition, and returns control back to the main graphics processor.

call *arg1.* This command causes the graphics processor to fetch graphics commands from the internal buffer of the previously defined figure number *arg1.*

line *arg1 arg2 arg3 arg4 arg5 arg6.* This command causes a line to be drawn in three-dimensional space from the point *arg1, arg2, arg3* to the point *arg4, arg5, arg6.* The current object rotation, object scale, object translation, viewer location, window, and viewport are used.

plot *arg1 arg2 arg3 arg4.* This command causes a line to be drawn from the endpoint of the last line plotted to the point *arg1, arg2, arg3* using the "pencode" *arg4.* The current pencodes supported are '2' (indicating that a solid line is to be drawn), and '3' (indicating that no line is to be drawn; this is used only to change the location of the plot head). Additional pencodes could be implemented allowing for dashed lines, dotted lines, etc.

ident. This command causes the "current" matrix to be set to the identify matrix. This causes all rotates to be set to zero, all translates to be set to the origin, and all scales to be set to one.

push. This command causes the current matrix to be pushed onto a 10 location matrix stack. The current matrix is not altered.

pop. This command causes the matrix stack to be popped into the current matrix.

rotate *arg1 arg2 arg3.* This command causes the viewer's perception of the three-dimensional figure to be rotated around the X, Y, and Z axis by *arg1, arg2* and *arg3.* The angles are in degrees. The definition of an object is not altered.

translate *arg1 arg2 arg3.* This command causes the viewer's perception of the three-dimensional figure to be translated in the X, Y, and Z directions by *arg1, arg2* and *arg3.* Again, the definition of an object is not altered.

scale *arg1 arg2 arg3.* This command causes the viewer's perception of the three-dimensional figure to be scaled in the X, Y and Z directions by *arg1, arg2,* and *arg3.*

window *arg1 arg2.* This command sets up the window parameters. These parameters determine the visible side to side portion of the projected images. This amounts to placing an infinitely tall pyramid within three-dimensional space with the viewing location located at its apex (looking down). All objects within this pyramid will be visible; all objects outside this pyramid will not be visible.

viewport *arg1 arg2 arg3 arg4.* This command sets up the viewport parameters. These parameters determine the size and location of the above window on the plotter surface. The center of the area on the plotter surface is given by *arg1, arg2* with the X and Y half sizes given by *arg3, arg4.* The plotter is assumed to have an X dimension between 0 and 12, and a Y dimension between 0 and 10. The translation to the dimensions the plotter recognizes is done in a lower level plotter interface routine. By performing this task in a lower level of software, the package is made more general.

viewpoint *arg1 arg2 arg3 arg4 arg5 arg6.* This command sets up the "viewing" transformation. *arg1, arg2, arg3* represent the location of the viewer in three-dimensional space, while *arg4, arg5, arg6* represent the "lookat" location in three-dimensional space. Together they form a vector pointing to the area to be viewed whose length determines the perspective variables (only single point perspective is currently implemented).

zclip *arg1 arg2.* This command sets up the "Z-clipping" parameters. These determine the visible distance in front of the viewer. *Arg1* specifies the near boundary of the viewing area while *arg2* specifies the far boundary of the area. Together with the window command, it defines a solid delimiting the visible objects from the not-visible objects.

cube *arg1 arg2 arg3 arg4 arg5 arg6.* This command draws a cube centered at *arg1, arg2, arg3* with half-widths of *arg4, arg5* and *arg6.*

arrow. This command draws an arrow from (0,0,0) to (1,0,0).

pyramid *arg1 arg2 arg3 arg4 arg5 arg6.* This command draws a four-sided pyramid whose base is centered at *arg1, arg2, arg3* and whose half-widths are *arg4, arg5, arg6.* The X half-width *arg4* is used as the height of the pyramid.

current. This command prints the current matrix on the terminal.

printdef. This command prints the definition of the given figure.

startt. This command starts the 10 ms timer on the iSBC 86/12 board.

readt. This command stops the 10 ms timer on the iSBC 86/12 board and prints the 10 ms count on the terminal.

end. This command stops execution of the graphics package, prints the total numbers of points plotted and "success!!!" on the terminal, and returns control back to ISIS.

Internal Operation of the Package

All internal operations are performed using 1 by 4 or 4 by 4 double precision real matrices. Points are defined in 1 by 4 double precision vectors where the first three coordinates are used to hold the X, Y and Z location of the point. The fourth location is always set to one, and is used when the point is projected onto a two-dimensional plane. In most cases, the routine performing the task outlined is named the same thing as the name of the task outlined (within the six-character limit imposed by FORTRAN). The order the routines are described is roughly the order a line would encounter them on its way from existing as a three-dimensional entity inside the machine to a line drawn on the bed of a plotter. All routine names are set in **boldface.**

THE CURRENT TRANSFORMATION

If each object were to be modified whenever a translate, rotate, or scale were to be performed, performance of the package could be quite slow. In addition, the original definition of the figure would be lost (although not irreversibly). If there were a method of performing these three operations at a single time, allowing the original definition of an object to remain unaltered, both the performance and ease of use of the graphics package would be enhanced.

One way in which these operations can be combined is by using what is called the "current" matrix. The current matrix is a 4 by 4 double precision real matrix. It numerically represents any combination of rotates, translates and scales in any order. The matrix is multiplied by each 1 by 4 point definition vector on its way to being plotted. The result of this multiplication is a point that has been rotated, scaled, and translated the proper amount. If this matrix is the identity matrix, the point will pass through unaltered. Thus, the identity matrix represents no scaling, translating, and rotating. This multiplication is performed in the routine **pline** lines 20 and 21.

When a rotate, scale, or translate command is interpreted, the current matrix is multiplied by another 4 by 4 matrix representing only this transformation. Since matrix multiplication is not commutative, the order these operations are performed in is preserved. This is important, because, for example, a rotate before a translate is not the same as a rotate after a translate because all rotations are performed pivoting around the origin (see Figure 16). Initially, the current matrix is set to the identity matrix. The first operation is performed relative to state of the current matrix immediately preceding the operation.

Parameters are set up into the current matrix through the rotate, scale, translate, ident, push, and pop operations. Each name describes the function of the operation performed. The routines performing these tasks (in order) are: **rotate, scale, transl, ident, push,** and **pop. Ident** is included to allow all rotates and translates to be set to zero and all scales to be set to one. The **push** and **pop** operations are included in order that figures may save the state of the current matrix, while subsequently performing operations altering it. This is important when a large figure is defined as a set of parts, each of which may merely be rotations, etc., of a simpler list of parts.

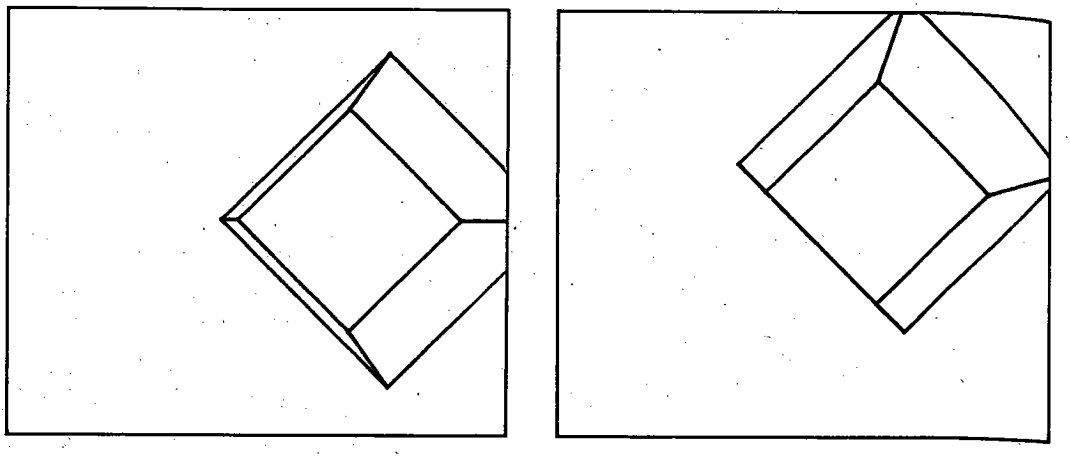

Figure 16. Example Cube Viewed from (0,0,10) First Rotated then Translated then Translated then Rotated.

Before an object can be plotted, the viewpoint of the viewer must be known. This information provides the location of the viewer in three-dimensional space, and the direction the viewer is pointing. It is incorporated into the 4 by 4 "view" matrix. It is another rotation performed on the object in order that it is viewed from the proper viewing angle. All points are passed through the view matrix after they are passed through the current matrix. What comes out of these two transformations is a set of points located in the proper relative positions in three-dimensional space when the figure is rotated, translated, and scaled by the operations performed on the current matrix, and is also rotated properly by the operations set in the view matrix.

The view matrix is set up by the viewpoint command. This command will place in the view matrix the proper rotations in order that the image of the object will be correct. The routine performing this task is the **viewpn** routine.

Z-CLIPPING

All points passed through the current and view matrices are located at their proper locations in three-dimensional space. However, only a portion of this space is visible to the viewer. Specifically, objects behind the viewer will not be visible. Every point of an object has been mapped to the viewer's space, however, including those behind the viewer. These "invisible" points are removed by an operation called "Z-clipping." Simply, it examines the Z parameter of every point being considered and determines if it is in front of the viewer. In addition, one may not wish to display lines a great distance from the viewer. These lines may be removed by a similar process. The only complication of clipping is the action performed if only part of the line is visible. In this instance, the point where the line leaves the visible area must be calculated. The method used to calculate this point in this implementation is the method of "like triangles."

The Z-clipping parameters are set through the command zclip in the routine **zclip.** The arguments to this command are used to determine the visible distance in front of the viewer. The first argument sets the minimum distance in front of the viewer before any line will be visible. Legal values for this parameter are anything greater than zero. The second argument sets the far distance beyond which no lines will be visible. Any value larger than the first argument may be used for this parameter. The clipping itself is performed in the routine **zclipp.**

PROJECTION

Projection maps the three-dimensional points previously encountered and projects them onto a two-dimensional plane. Only single-point perspective is currently supported in the package. Here, the projection is performed by using the Z parameter to modify the X and Y parameters. As the points get more distant, their deviation from the center of the picture should get smaller, if the X and Y parameters remain constant. Most people are aware of this effect. For example, if you look down a set of railroad tracks, the rails seem to converge, even though the distance between the rails is constant (see Figure 17). Two or three-point perspective would be easy to implement; all one must do is generate the projected X and Y parameters by using the non-projected X and Y parameters in addition to using the Z parameter.

This projection is performed in the graphics package by multiplying the 1 by 4 point location vector by a 4 by 4 "projection" matrix. This matrix is simply the identity matrix except the perspective value is placed in location (3,4) of the matrix.

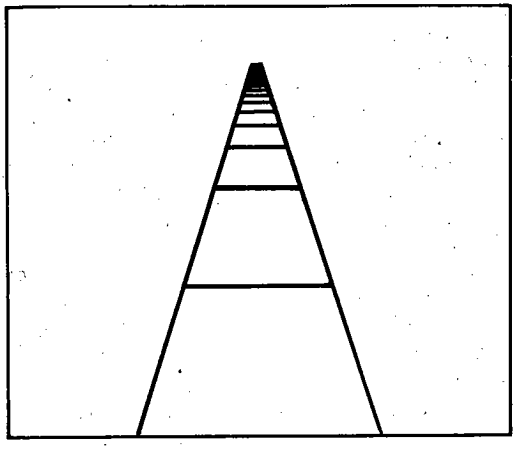

Figure 17. Two Rails, Vanishing into the Distance

This value is calculated from the viewpoint parameters. After the matrix multiply, the only element modified in the 1 by 4 point definition vector is the last one (the one which is supposed to have the value of one). After the multiplication, this location will contain the number representing the modification which must be performed on the X and Y parameters of the vector to exhibit the projection. When this vector is "normalized," the point will have been projected using the rules of single-point perspective. This normalization is performed by dividing every element in the vector by the last element of the vector. Thus, the Z element of the original vector has modified the X and Y elements. If two or three-point perspective is desired, one must only place perspective values in locations (1,4) and (2,4) of the projection matrix; all subsequent processing will be identical. The routines performing these operations are: **viewpn** (sets up vanishing point for perspective), **project** (sets up the projection matrix, and performs the perspective multiplication), and **norm** (normalizes the vector).

X-Y CLIPPING

Once the data is projected onto a two-dimensional plane, X-Y clipping must be performed. This operation could also be performed on the three-dimensional data, but by deferring it until after the data have been projected, the calculations required are simpler. This is not true for Z-clipping, since once the data are projected onto a plane, the Z parameter is no longer in its original form.

X-Y clipping is performed by comparing X and Y parameters with the window values set up by the window command. This comparison is a bit more complicated than the comparison required by Z clipping, however, as two clipping parameters are involved. There are nine possible regions in which each endpoint of a line may reside. For example, some of these regions are: within the X and Y window regions, less than the X window region but within the Y region, less than the X region and less than the Y region, etc. If one or both of the endpoints of the line are within the visible region, then at least part of the line will be visible. Also, even if neither of the endpoints of the line is in the visible region, part of the line may still be visible. One must therefore determine whether any part of this line would be visible. A simple way of performing the task is to assign a bit of a word for each of less than and greater than the X and Y window limits. This requires four bits. The value of the X and Y parameters are then each compared with the window limits. If the value exceeds the limit of the window, the corresponding bit of this point descriptor is set. After this "code" has been determined for both of the points, the codes for two endpoints are bit-wise ANDed together (an extension to FORTRAN 77 available in FORTRAN 86 allows this operation). If the result of this ANDing is zero, then part of the line would be visible. If, however, it is not zero, then the entire line lies outside the visible area. If only part of the line is visible, then the point where it leaves the visible area must be calculated. The point where the line leaves the viewing area is calculated using the same "like triangle" method used when Z-clipping is performed.

The routines performing these operations are **wtovp** (calls the **xyclip** routine with the proper parameters), **xyclip** (performs the actual clipping), **code** (returns the binary code for the point position in relation to the window), and **ppush** (calculates the point at which line leaves the visible area).

WINDOW TO VIEWPORT TRANSFORMATION

Finally, after the points have been processed through all of the above, comes their day of glory. Because the lines have been clipped, they are constrained to be within the given window. Remember, however, that the values for this window are in "real world" units. These sizes could be measured in inches or miles. These are not generally suitable for plotting on a graphics output device. In order for the "window" to be displayed on the graphics output device, one more transformation must be performed: the window to viewport transformation. A viewport represents a physical location and size on the graphics output device. The viewport command sets up the appropriate parameters for this transformation. It requires four arguments, which allow the viewport to be moved around the graphics display surface, and allow the size of the viewport to be set. Notice that the viewport and the window are not constrained to the same aspect ratios, that is, the ratios between the vertical sizes and the horizontal sizes of the window and viewport need not be the same. If these ratios are not the same, the figures will be distorted. Performing this transformation is simply a matter of scaling the windowed values to fill the viewport. The code performing this transformation is contained within the **wtovp** routine.

PLOTTER INTERFACE

This graphics package was written to interface to a Hewlett-Packard 7225A flat bed plotter. Communications were performed through an RS232 serial link at 600 baud. Physically, this is done using the 8251 serial controller on the iSBC® 86/12 board inside the Intellec® Series III. The plotter has a smart interface. The commands it accepts are in ASCII, and are on the level of "lower the pen," and "draw a line from the current pen position to another pen position." The routines performing these operations are **plot** (determines the characters needing to be sent to the plotter), **ponum** (converts a floating point number to an ASCII representation of the integer value of the truncated floating point number), **putout** (handles the interface to the 8251 serial controller chip) and **plots** (initializes the baud rate generator and 8251 serial controller chip on the iSBC® 86/12 board).

PERFORMANCE MEASUREMENTS

The above routines were compiled using the Intel FORTRAN 86 compiler and exeucted on an Intellec® Series III development system. The 8086 hardware consists of an Intel iSBC® 86/12 board with the 8087 in the iSBC® 337 card. The iAPX 86/20 (the 8086 with the 8087) operate with a clock frequency of 5 MHz. The on board memory (64K DRAM) inserts between one and three wait states per memory fetch. In addition, owing to the size of the memory arrays, the program size, and the memory reguirements of the Series III, off board memory was required to run the program.

The times shown in the table do not show the plotting time; only the time to generate the output that would be sent to the plotter is given. This is because the physical speed limitation of the plotter used would not allow the iAPX 86/20 system to produce the plotting commands at its maximum computational speed. The plotter required approximately half an hour to 45 minutes to actually draw the second demonstration picture.

For each line plotted, five 1 by 4 times 4 by 4 matrix multiplies must be performed along with a non-trivial amount of other floating point operations, such as divides and compares. For example, when clipping is performed, the line endpoint values must be compared to the clipping parameters. If only part of the line is visible, then the point the line leaves the visible area must be calculated. This requires twelve additional floating point operations. Another example is in the window to viewport transformation. For each line drawn, four floating point multiplies, four floating point divides, and four floating point adds must be performed.

In addition, whenever the rotation, scale, translation or viewpoint is changed, 4 by 4 matrix multiplies must be performed. In addition, various trigonometric routines, such as sines and cosines, must be performed to set up the rotation parameters into the matrix.

The performance measurements are given in Table 1.

Table 1. Performance Measurements

	Picture Number	
	One	Two
number of points in picture	117	9131
number of points actually plotted	117	6114
execution time of the 86/20(sec)	2.84	188
execution time of the 86 with 87 emulator(sec)	144.77	9801
exection time of PDP11/45(sec)[2]	1.7	120

[2]A PDP11/45 mini-computer with 256K MOS RAM, and a FP11-B floating point unit running the UNIX operating system during a period of light load. The program was compiled using the UNIX F77 FORTRAN compiler.

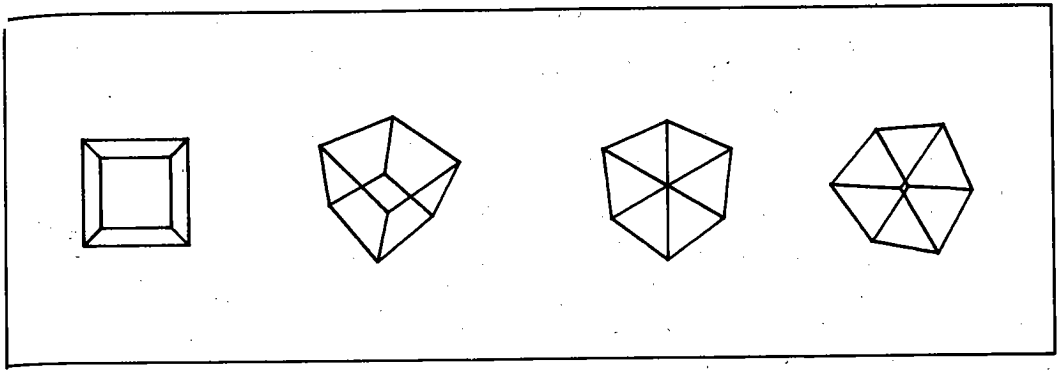

Figure 18. Demonstration Picture 1

The results show that the performance of the iAPX 86/20 is close to the performance of the mini-computer. The figures drawn are shown in Figure 17 for Picture 1 and Figure 18 for Picture 2. The graphics commands required to generate Picture 1 are given in Appendix B. Picture 2 shows three views of a single shuttle. (Hint: you are looking out the window of one of the shuttles!) The shuttle is defined only once in the input data. Another point to notice is that each shuttle is a conglomeration of parts. For example, the shuttle wing is defined only once in input data. The complete shuttle contains two views of this same wing, translated and rotated to attach to the appropriate location on the fuselage of the shuttle itself. The engine nozzles take this same approach a bit further. The complete nozzle is defined only once, and is attached in three places on each shuttle. In addition, each nozzle is made up of replications of the same circle scaled and translated through space. Each circle is, in turn, composed of four views of one quarter-circle, each rotated a proper amount to form one complete circle.

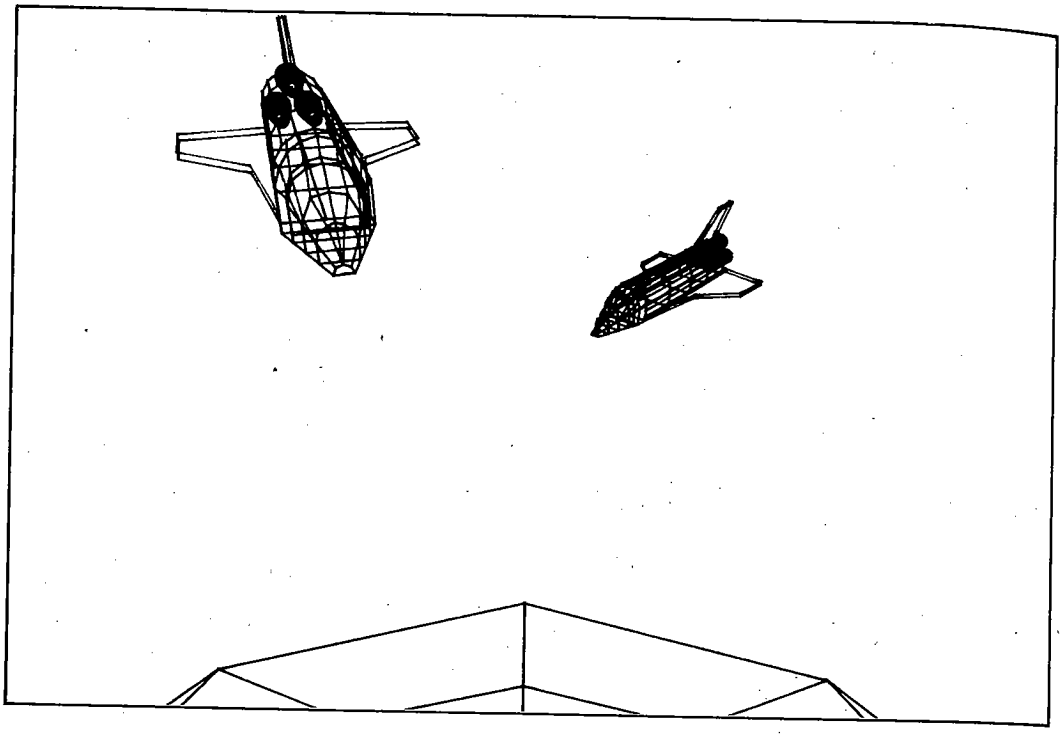

Figure 19. Demonstration Picture 2

CONCLUSIONS

The routines demonstrated in this note show that the types of operations required to manipulate and display a three-dimensional figure on a two-dimensional surface are far from trivial, involving very many floating point operations. With the introduction of the iAPX 86/20, the floating point performance required by this type of application is finally within the performance limits of microcomputers selling for a fraction of the cost of the previously required mini- or maxi-computers. Examples of systems in which this performance is required are Computer Aided Design (CAD) or Computer Aided Manufacturing (CAM) systems. In addition, the availability of a full ANSI 77 standard FORTRAN compiler (FORTRAN 86) for the iAPX 86/20 enhances the production or transportation of existing software to the machine. This combination of high performance hardware with high performance software allows the iAPX 86/20 to fill applications never before filled by a microprocessor.

APPENDIX A Contents

```
      c
      c this is the main routine of the graphics program.  basically
      c  it sets up default parameters for the rest of the routines, then
      c  enters an infinite loop, alternatively fetching lines from the input
      c  (using routine getl) and sending them to be processed by the graphics
      c  processor (proc)
      c
 1            common /windoe/wxh,wyh
 2            common /viewp/vxh,vyh,vxc,vyc
 3            real*8 wxh,wyh,vxh,vyh,vxc,vyc
 4            common /matrix/currm,view,curp
 5            real*8 currm(4,4),view(4,4),curp(4)
 6            common /clip/hither,yon,dee
 7            real*8 hither,yon,dee
 8            common /stacks/stackp,sspace
 9            real*8 sspace(10,4,4)
10            integer stackp
11            common /defns/darg1,darg2,darg3,darg4,darg5,darg6,darg7,entry,tailp,ends
12            character*10 darg1(500)
13            real*8 darg2(500),darg3(500),darg4(500),darg5(500),darg6(500),darg7(500)
14            integer entry(10),ends(10)
15            integer tailp
16            common /cstack/cnum,cnump
17            integer cnum(10),cnump
18            common /penpos/xpos,ypos,pcount
19            real*8 xpos,ypos
20            integer*4 pcount

      c  initialize the plotting package
21            call plots
      c       initialize the stack pointer
22            stackp = 1

      c  set up a few defaults
23            wxh = 10.
24            wyh = 10.
25            vxh = 5.
26            vyh = 5.
27            vxc = 5.
28            vyc = 5.
29            hither = 1.
30            yon = 100.
31            dee = 10.
32            tailp = 1
33            cnump = 1
34            xpos = -1.
35            ypos = -1.
36            pcount = 0
37            print *,'GRAPHICS program entered!!!'
      c
      c  initialize the current matrix
      c
38            call ident(currm)
      c
      c and process all the input lines
      c
39    100             call getl
40                    call proc
41            goto 100
42            end
```

```
c
c       getl(line)
c
c       fetches the next line from the input file, and grabs the first 7
c       things from it, the first being an alpha command contained within
c       (') and the rest being numbers.  If less than 6 number are input
c       the input line must be terminated by a (/) in order for the
c       read statement to be correctly interpreted.  The arguments are then
c               placed in the common block "args".  When the 'end' command is
c               encountered, "success" is printed on the terminal, and the
c               graphics program terminates.
c
43          subroutine getl
44          common /args/arg1,arg2,arg3,arg4,arg5,arg6,arg7
45          character*10 arg1
46          real*8 arg2,arg3,arg4,arg5,arg6,arg7

47          read (5,*)arg1,arg2,arg3,arg4,arg5,arg6,arg7
48          if(arg1 .eq. 'end') then
49                  call plot(0.,0.,999)
50                  print *,'success!!!'
51                  stop
52          endif
53          return
54          end


c
c       proc
c
c       proc() does all the processing for a line.  It gets its arguments
c       from the common block args, and does it's thing
c
55          subroutine proc

56          common /matrix/currm,view,curp
57          real*8 currm(4,4),view(4,4),curp(4)
58          common /args/arg1,arg2,arg3,arg4,arg5,arg6,arg7
59          character*10 arg1
60          real*8 arg2,arg3,arg4,arg5,arg6,arg7
61          common /clip/hither,yon,dee
62          real*8 hither,yon,dee
63          common /cstack/cnum,cnump
64          integer cnum(10),cnump
65          integer i
66          integer*4 rtimer,countt

c
c               determine the command entered (HUGE if-then-else if-,etc) and
c               call the appropriate routine with the correct arguments
c
67          if(arg1 .eq. 'comment') then
68                  i = 1
69  100             read(5,800)
70                  i = i + 1
71                  if(i .le. int(arg2)) goto 100
72  800             format(a1)
73          else if(arg1 .eq. 'define') then
74                  i = int(arg2)
75                  call defn(i)
76                  call printd(i)
77          else if(arg1 .eq. 'call') then
78                  cnum(cnump) = int(arg2)
79                  cnump = cnump + 1
80                  if(cnump .gt. 10) then
81                          print *,'call nesting level too deep, sorry'
82                          cnump = 10
83                  endif
84                  call callit(cnum(cnump - 1),cnump - 1)
85                  cnump = cnump - 1
86          else if(arg1 .eq. 'line') then
```

```
                call pline(arg2,arg3,arg4,arg5,arg6,arg7,2)
        else if(arg1 .eq. 'plot') then
                i = int(arg5)
                call pplot(arg2,arg3,arg4,i)
        else if(arg1 .eq. 'ident') then
                call ident(currm)
        else if(arg1 .eq. 'push') then
                call push(currm)
        else if(arg1 .eq. 'pop') then
                call pop(currm)
        else if(arg1 .eq. 'rotate') then
                call rotate(arg2,arg3,arg4,currm)
        else if(arg1 .eq. 'translate') then
                call transl(arg2,arg3,arg4,currm)
        else if(arg1 .eq. 'scale') then
                call pscale(arg2,arg3,arg4,currm)
        else if(arg1 .eq. 'window') then
                call window(arg2,arg3)
        else if(arg1 .eq. 'viewport') then
                call viewpr(arg2,arg3,arg4,arg5)
        else if(arg1 .eq. 'viewpoint') then
                call viewpn(arg2,arg3,arg4,arg5,arg6,arg7)
        else if(arg1 .eq. 'zclip') then
                call zclip(arg2,arg3)
        else if(arg1 .eq. 'cube') then
                call cube(arg2,arg3,arg4,arg5,arg6,arg7)
        else if(arg1 .eq. 'arrow') then
                call arrow
        else if(arg1 .eq. 'pyramid') then
                call pyrmd(arg2,arg3,arg4,arg5,arg6,arg7)
        else if(arg1 .eq. 'current') then
                call printm(currm)
        else if(arg1 .eq. 'printdef') then
                i = int(arg2)
                call printd(i)
        else if(arg1 .eq. 'startt') then
                call stimer
        else if(arg1 .eq. 'readt') then
                countt = rtimer()
                print *,'the time (in seconds) from the last startt is:',countt/100
        else
                print *,'error, command ',arg1,'unknown'
        endif

        return
        end

c
c       ident(matrx)
c
c       ident() sets the given 4 X 4 matrix to the identity matrix.
c
        subroutine ident(matrx)
        real*8 matrx(4,4)
        integer i,j

        do 100 i=1,4
                do 100 j=1,4
                        matrx(i,j) = 0.
100     continue
        do 110 i=1,4
                matrx(i,i) = 1.
110     continue
        return
        end
```

```
c
c       subroutine defn(number) defines figure number.  the defined figure
c               is contained in a large common block "defns" which contains
c               enough space for a total of 500 commands.  comments are not
c               stored along with the define commands to save space.  the variables
c               entry and ends contain the starting and ending indexes of the
c               10 possible defined figures
c
144             subroutine defn(number)
145             integer number
146             common /defns/darg1,darg2,darg3,darg4,darg5,darg6,darg7,entry,tailp,ends
147             character*10 darg1(500)
148             real*8 darg2(500),darg3(500),darg4(500),darg5(500),darg6(500),darg7(500)
149             integer entry(10),ends(10)
150             integer tailp
151             common /args/arg1,arg2,arg3,arg4,arg5,arg6,arg7
152             character*10 arg1
153             real*8 arg2,arg3,arg4,arg5,arg6,arg7
154             integer i

155             entry(number) = tailp
156             print *,'start of define is at',tailp

157     100     call getl
        c
        c       check for terminate of define
        c
158             if(arg1 .eq. 'enddef') then
159                     ends(number) = tailp
160                     print *,'end of figure define is at',tailp
161                     return
162             else if(arg1 .ne. 'comment') then
163                     darg1(tailp) = arg1
164                     darg2(tailp) = arg2
165                     darg3(tailp) = arg3
166                     darg4(tailp) = arg4
167                     darg5(tailp) = arg5
168                     darg6(tailp) = arg6
169                     darg7(tailp) = arg7
170                     tailp = tailp + 1
171                     if(tailp .gt. 500) then
172                             print *,'define memory overrun!!!'
173                             tailp = 500
174                     endif
175             else
176                     i = 1
177     150             read(5,800)
178                     i = i + 1
179                     if(i .le. int(arg2)) goto 150
180     800             format(a1)
181             endif
182             goto 100
183             end
```

```
      c
      c       subroutine printd(number) prints the defined figure commands
      c
184           subroutine printd(number)
185           integer number
186           common /defns/darg1,darg2,darg3,darg4,darg5,darg6,darg7,entry,tailp,ends
187           character*10 darg1(500)
188           real*8 darg2(500),darg3(500),darg4(500),darg5(500),darg6(500),darg7(500)
189           integer entry(10),ends(10)
190           integer tailp
191           integer i

192           i = entry(number)
193     100   if(i .eq. ends(number)) return
194           write(6,800)darg1(i),darg2(i),darg3(i),darg4(i),darg5(i),darg6(i),darg7(i)
195     800   format(a10,6f11.4)
196           i = i + 1
197           goto 100
198           end


      c
      c       subroutine callit(number,nest)  causes the defined figure number to
      c               be input to the graphics processor, nesting level must be provided
      c               to allow pseudo-recursive type calls...
      c
199           subroutine callit(number,nest)
200           integer number,nest
201           common /defns/darg1,darg2,darg3,darg4,darg5,darg6,darg7,entry,tailp,ends
202           character*10 darg1(500)
203           real*8 darg2(500),darg3(500),darg4(500),darg5(500),darg6(500),darg7(500)
204           integer entry(10),ends(10)
205           integer tailp
206           common /args/arg1,arg2,arg3,arg4,arg5,arg6,arg7
207           character*10 arg1
208           real*8 arg2,arg3,arg4,arg5,arg6,arg7
209           integer i(10)

210           i(nest) = entry(number)
211     100   if(i(nest) .eq. ends(number)) return
212           arg1 = darg1(i(nest))
213           arg2 = darg2(i(nest))
214           arg3 = darg3(i(nest))
215           arg4 = darg4(i(nest))
216           arg5 = darg5(i(nest))
217           arg6 = darg6(i(nest))
218           arg7 = darg7(i(nest))
219           call proc
220           i(nest) = i(nest) + 1
221           goto 100
222           end


      c
      c       printm(matrx)
      c
      c       printm prints out the given 4x4 double precision matrix
      c
223           subroutine printm(matrx)
224           real*8 matrx(4,4)
225           integer i

226           do 100 i=1,4
227                   write(6,800)matrx(i,1),matrx(i,2),matrx(i,3),matrx(i,4)
228     100   continue
229     800   format(4f15.4)
230           return
231           end
```

```
      c
      c       pline(x,y,z,a,b,c,s)
      c
      c       pline() draws a line from (x,y,z) to (a,b,c) with pencode s, using
      c       the current window, viewpoint, viewport, etc.
      c
 1            subroutine pline(x,y,z,a,b,c,s)
 2            real*8 x,y,z,a,b,c
 3            integer s
 4            common /matrix/currm,view,curp
 5            real*8 currm(4,4),view(4,4),curp(4)
 6            logical zclipp,junk
 7            real*8 tmpf(4),tmpt(4)

 8            tmpf(1) = x
 9            tmpf(2) = y
10            tmpf(3) = z
11            tmpf(4) = 1.
12            tmpt(1) = a
13            tmpt(2) = b
14            tmpt(3) = c
15            tmpt(4) = 1.
16            curp(1) = a
17            curp(2) = b
18            curp(3) = c
19            curp(4) = 1.
      c
      c  perform translations, and viewing translation
      c

20            call mmultl(tmpf,currm,tmpf)
21            call mmultl(tmpt,currm,tmpt)
22            call mmultl(tmpf,view,tmpf)
23            call mmultl(tmpt,view,tmpt)
      c
      c  perform zclipping on both points...
      c
24            if(zclipp(tmpf,tmpt).eq. .false.) goto 200
25            junk=zclipp(tmpt,tmpf)
      c
      c  project the vector into 2-D
      c
26            call projct(tmpf)
27            call projct(tmpt)
      c
      c  do x/y clipping, the window to viewport transform, and plot the vector
      c
28            call wtovp(tmpf,tmpt,s)
29    200     return
30            end
```

```
      c
      c       pplot(x,y,z,t)
      c
      c       plot a line from the current position to (x,y,z) using pencode t.
      c       Basically, sets up a call to pline from the current position
      c       to the new position using the appropriate pencode.
      c
31            subroutine pplot(x,y,z,t)
32            real*8 x,y,z
33            integer t
34            common /matrix/currm,view,curp
35            real*8 currm(4,4),view(4,4),curp(4)

36            call pline(curp(1),curp(2),curp(3),x,y,z,t)
37            return
38            end


      c
      c       push(matrix)
      c
      c       push() pushes the given matrix onto the matrix stack, checks
      c       for stack overflow, and won't let you!!!!  Does not alter the
      c       current matrix.
      c
39            subroutine push(matrix)
40            real*8 matrix,sspace(4,4,10)
41            integer stackp
42            dimension matrix(4,4)
43            common /stacks/stackp,sspace

44            if(stackp .gt. 10) then
45                    print *,'stack overflow'
46                    return
47            end if
48            call copym(sspace(1,1,stackp),matrix)
49            stackp=stackp+1
50            return
51            end


      c
      c       pop(matrix)
      c
      c       pop() pops the top of stack into the given matrix.  Checks for
      c       stack underflow, and again won't let you do it!!!!
      c
52            subroutine pop(matrix)
53            real*8 matrix,sspace(4,4,10)
54            integer stackp
55            dimension matrix(4,4)
56            common /stacks/stackp,sspace

57            stackp=stackp-1
58            if(stackp .lt. 1) then
59                  print *,'stack underflow'
60                  stackp = 1
61                  return
62            end if
63            call copym(matrix,sspace(1,1,stackp))
64            return
65            end
```

```
c
c       rotate(x,y,z,matrix)
c
c       rotate() pre-concatenates the given (x,y,z) rotation, to the
c       supplied matrix(usually the current matrix).  x,y,z are given
c               in degrees.
c
1       subroutine rotate(x,y,z,matrix)
2       real*8 x,y,z,matrix
3       dimension matrix(4,4)
4       real*8 tmp
5       dimension tmp(4,4)

6       call ident(tmp)
7       tmp(2,2) = cos(x * 0.01745329)
8       tmp(3,3) = tmp(2,2)
9       tmp(2,3) = sin(x * 0.01745329)
10      tmp(3,2) = - tmp(2,3)

11      call mmult4(tmp,matrix,matrix)

12      call ident(tmp)
13      tmp(1,1) = cos(y * 0.01745329)
14      tmp(3,3) = tmp(1,1)
15      tmp(3,1) = sin(y * 0.01745329)
16      tmp(1,3) = - tmp(3,1)

17      call mmult4(tmp,matrix,matrix)

18      call ident(tmp)
19      tmp(1,1) = cos(z * 0.01745329)
20      tmp(2,2) = tmp(1,1)
21      tmp(1,2) = sin(z * 0.01745329)
22      tmp(2,1) = - tmp(1,2)

23      call mmult4(tmp,matrix,matrix)

24      return
25      end


c
c       translate(x,y,z,matrix)
c
c       translate() pre-concatenates the given tranlation (x,y,z) to the
c       given matrix(usually the current matrix).
c
26      subroutine transl(x,y,z,matrix)
27      real*8 x,y,z,matrix
28      dimension matrix(4,4)

29      real*8 tmp
30      dimension tmp(4,4)

31      call ident(tmp)
32      tmp(4,1) = x
33      tmp(4,2) = y
34      tmp(4,3) = z

35      call mmult4(tmp,matrix,matrix)

36      return
37      end
```

```
c
c       pscale(x,y,z,matrix)
c
c       pscale pre-concatenates the given scaling (x,y,z) onto the
c       given matrix.
c
38          subroutine pscale(x,y,z,matrix)
39          real*8 x,y,z,matrix
40          dimension matrix(4,4)

41          real*8 tmp
42          dimension tmp(4,4)

43          call ident(tmp)
44          tmp(1,1) = x
45          tmp(2,2) = y
46          tmp(3,3) = z

47          call mmult4(tmp,matrix,matrix)

48          return
49          end

c
c       window(a,b) viewport(a,b,c,d)
c
c       these two routines set up the global variables according to the
c       given parameters.
c
50          subroutine window(a,b)
51          real*8 a,b
52          real*8 wxh,wyh
53          common /windoe/wxh,wyh

54          wxh = a
55          wyh = b
56          return
57          end

c
58          subroutine viewpr(a,b,c,d)
59          real*8 a,b,c,d
60          real*8 vxh,vyh,vxc,vyc
61          common /viewp/vxh,vyh,vxc,vyc

62          vxc = a
63          vyc = b
64          vxh = c
65          vyh = d
66          call mplot(vxc - vxh,vyc - vyh,3)
67          call mplot(vxc + vxh,vyc - vyh,2)
68          call mplot(vxc + vxh,vyc + vyh,2)
69          call mplot(vxc - vxh,vyc + vyh,2)
70          call mplot(vxc - vxh,vyc - vyh,2)
71          return
72          end
```

```
      c
      c       viewpoint(a,b,c,d,e,f)
      c
      c       viewpoint sets up the viewing transformation for the given
      c       to and from points---the eye position is (a,b,c) the lookat
      c       position is (d,e,f).
      c
 1            subroutine viewpn(a,b,c,d,e,f)
 2            real*8 a,b,c,d,e,f

 3            real*8 angle
 4            real*8 tmp(4,4),tmpp(4)
 5            common /matrix/currm,view,curp
 6            real*8 currm(4,4),view(4,4),curp(4)
 7            common /clip/hither,yon,dee
 8            real*8 hither,yon,dee

      c
      c                initialize the viewing transformation
      c
 9            call ident(view)
      c
      c                move lookat position to origin
      c
10            call transl(-d,-e,-f,view)
      c
      c                rotate view matrix per the lookat angle
      c
11            a = a - d
12            b = b - e
13            c = c - f
14            angle = - atan2(a,c)
15            call ident(tmp)
16            tmp(1,1) = cos(angle)
17            tmp(3,3) = tmp(1,1)
18            tmp(3,1) = sin(angle)
19            tmp(1,3) = - tmp(3,1)

20            call mmult4(view,tmp,view)

21            angle = atan2(b,sqrt(a*a + c*c))
22            call ident(tmp)
23            tmp(2,2) = cos(angle)
24            tmp(3,3) = tmp(2,2)
25            tmp(2,3) = sin(angle)
26            tmp(3,2) = - tmp(2,3)

27            call mmult4(view,tmp,view)

28            a = a + d
29            b = b + e
30            c = c + f
31            tmpp(1) = a
32            tmpp(2) = b
33            tmpp(3) = c
34            tmpp(4) = 1.

35            call mmult1(tmpp,view,tmpp)

36            dee = tmpp(3)
37            return
38            end
```

```
c
c       zclip(a,b)
c
c       zclip() sets up the global clipping parameters, a is the hither,
c       b the yon, does not allow the hither plane to be behind the
c       viewer, nor does it allow the yon to be between the viewer
c       and the hither.
c
39          subroutine zclip(a,b)
40          real*8 a,b
41          real*8 hither,yon,dee
42          common /clip/hither,yon,dee

43          if(a .lt. 0) then
44             a     print *,'bad hither parameter'
45                   a = 0
46          end if
47          if(b .lt. a) then
48                   print *,'bad yon parameter'
49                   b = a + 100
50          end if
51          hither = a
52          yon = b
53          return
54          end


c
c       zclipping(vect1,vect2)
c
c       zclipping() performs the zclipping on vect1 using the global
c       zclipping parameters.  Modifies ONLY vect1, returns true if
c       a portion of the vector indicated by (clipped)vect1 and vect2
c       will be visible in the scene.
c
55          logical function zclipp(vect1,vect2)
56          real*8 vect1(4),vect2(4)
57          common /clip/hither,yon,dee
58          real*8 hither,yon,dee

59          real*8 htr,yn

60          htr = dee - hither
61          yn = dee - yon

62          zclipp = .true.

63          if(vect1(3) .gt. htr) then
64                  if(vect2(3) .gt. htr) then
65                          zclipp = .false.
66                  else
c
c       you must modify the x and y parameters (according to like triangles)
c       when the z parameter is modified!!!
c
67                          vect1(1) = (vect1(1) - vect2(1))*((htr - vect2(3))/
     *                                  (vect1(3) - vect2(3))) + vect2(1)
68                          vect1(2) = (vect1(2) - vect2(2))*((htr - vect2(3))/
     *                                  (vect1(3) - vect2(3))) + vect2(2)
69                          vect1(3) = htr
70                          zclipp = .true.
71                  end if
72          else if(vect1(3) .lt. yn) then
73                  if(vect2(3) .lt. yn) then
74                          zclipp = .false.
75                  else
```

```
76                          vectl(1) = (vect2(1) - vectl(1))*((yn - vectl(3))/
       *                                (vect2(3) - vectl(3))) + vectl(1)
77                          vectl(2) = (vect2(2) - vectl(2))*((yn - vectl(3))/
       *                                (vect2(3) - vectl(3))) + vectl(2)
78                          vectl(3) = yn
79                          zclipp = .true.
80                  end if
81              end if
82              return
83              end


    c
    c       projct(vector)
    c
    c       projct() projects the given vector to a point in 2-D space using
    c       the global "dee" parameter, for single point perspective.
    c
 1          subroutine projct(vector)
 2          real*8 vector(4)
 3          common /clip/hither,yon,dee
 4          real*8 hither,yon,dee
 5          real*8 tmp(4,4)

 6          call ident(tmp)

 7          if(dee .ne. 0) then
 8                  tmp(3,4) = - 1 / dee
 9          else
10                  tmp(3,4) = - 1000000000.
11          endif

12          call mmultl(vector,tmp,vector)
13          call norm(vector)
14          return
15          end


    c
    c       norm(vector)
    c
    c       norm() normalizes the given vector.
    c
    c
16          subroutine norm(vector)
17          real*8 vector(4)

18          vector(1) = vector(1) / vector(4)
19          vector(2) = vector(2) / vector(4)
20          vector(3) = vector(3) / vector(4)
21          vector(4) = 1.
22          return
23          end
```

```
      c
      c       wtovp(from,to,pencode)
      c
      c       wtovp() takes the projected from and to points, and:
      c               1: does x/y clipping on the window
      c               2: does the window to viewport translation
      c               3: plots the transformed points onto the device
      c
24            subroutine wtovp(from,to,pencde)
25            real*8 from(4),to(4)
26            integer pencde
27            common /windoe/wxh,wyh
28            real*8 wxh,wyh
29            common /viewp/vxh,vyh,vxc,vyc
30            real*8 vxh,vyh,vxc,vyc
31            logical xyclip
32            real*8 xp,yp

33            if(xyclip(from,to)) then
34                    xp = (from(1)) * vxh / wxh + vxc
35                    yp = (from(2)) * vyh / wyh + vyc

36                    call mplot(xp,yp,3)
37                    xp = (to(1)) * vxh / wxh + vxc
38                    yp = (to(2)) * vyh / wyh + vyc

39                    call mplot(xp,yp,pencde)
40            endif
41            return
42            end

      c
      c       xyclip(from,to)
      c
      c       xyclip() performs the x/y clipping on both the from and t
      c       vectors in the window cooridinates.  Returnes false if
      c       none of the vector would be visible.
      c
43            logical function xyclip(from,to)
44            real*8 from(4),to(4)
45            integer*2 cf,ct

46            xyclip = .false.
47    100             call code(from,cf)
48                    call code(to,ct)
49            if((cf .and. ct) .ne. 0) goto 105

50                    if(cf .ne. 0) call ppush(cf,from,to)
51                    if(ct .ne. 0) call ppush(ct,to,from)
52            if((cf + ct) .ne. 0) goto 100
53            xyclip = .true.

54    105     return
55            end
```

```
c
c       code(vector,flag)
c
c       code() returns the binary code in flag for vector indicating
c       it's position relative to the window.
c
1       subroutine code(vector,flag)
2       real*8 vector(4)
3       integer flag
4       common /windoe/wxh,wyh
5       real*8 wxh,wyh
6       real*8 tmp

7       flag = 0

8       tmp = vector(1)
9       if(tmp .lt. - wxh) flag = 1
10      if(tmp .gt. wxh) flag = flag + 2
11      tmp = vector(2)
12      if(tmp .lt. -wyh) flag = flag + 4
13      if(tmp .gt. wyh) flag = flag + 8
14      return
15      end


c
c       ppush(flag,to,from)
c
c       ppush() pushes "to" towards "from" according to flag, which
c       contains the code returned by code().  used to insure that the
c              line exits the window at the correct point
c
16      subroutine ppush(flag,to,from)
17      real*8 to(4),from(4)
18      integer flag
19      common /windoe/wxh,wyh
20      real*8 wxh,wyh

21        if((flag .and. 1) .ne. 0) then

22                to(2) = ((-wxh - from(1))
     *                  /(to(1) - from(1)))*(to(2) - from(2)) + from(2)
23                to(1) = -wxh
24        endif
25        if((flag .and. 2) .ne. 0) then

26                to(2) = ((wxh - from(1))
     *                  /(to(1) - from(1)))*(to(2) - from(2)) + from(2)
27                to(1) = wxh
28        endif
29        if((flag .and. 4) .ne. 0) then

30                to(1) = ((-wyh - from(2))
     *                  /(to(2) - from(2)))*(to(1) - from(1)) + from(1)
31                to(2) = -wyh
32        endif
33        if((flag .and. 8) .ne. 0) then

34                to(1) = ((wyh - from(2))
     *                  /(to(2) - from(2)))*(to(1) - from(1)) + from(1)
35                to(2) = wyh
36        endif
37        return
38      end
```

```
      c
      c       copym(dst,src)
      c
      c       copym() copies the src 4X4 matrix to the dst 4X4 matrix.
      c
 39           subroutine copym(dst,src)
 40           real*8 dst(16),src(16)

 41           integer i

 42           do 100 i = 1,16
 43           dst(i) = src(i)
 44    100    continue
 45           return
 46           end


      c
      c       mplot(arg1,arg2,arg3)
      c
      c       mplot() calls plot with arg1,arg2,arg3.  inserted as another level
      c               of indirection in order to allow the actual plot commands to be
      c               written to a file, etc.
      c
 47           subroutine mplot(arg1,arg2,arg3)
 48           real*8 arg1,arg2
 49           integer arg3

 50           call plot(arg1,arg2,arg3)
 51           return
 52           end


      c
      c       cube(arg1,arg2,arg3,arg4,arg5,arg6)
      c
      c       cube() generates a cube centered at (arg1,arg2,arg3) with
      c       arg4,arg5,arg6 as it's half widths
      c
  1           subroutine cube(arg1,arg2,arg3,arg4,arg5,arg6)
  2           real*8 arg1,arg2,arg3,arg4,arg5,arg6

  3           call pline(arg1-arg4,arg2-arg5,arg3-arg6,arg1+arg4,arg2-arg5,arg3-arg6,2)
  4           call pplot(arg1+arg4,arg2+arg5,arg3-arg6,2)
  5           call pplot(arg1-arg4,arg2+arg5,arg3-arg6,2)
  6           call pplot(arg1-arg4,arg2-arg5,arg3-arg6,2)
  7           call pplot(arg1-arg4,arg2-arg5,arg3+arg6,2)
  8           call pplot(arg1+arg4,arg2-arg5,arg3+arg6,2)
  9           call pplot(arg1+arg4,arg2+arg5,arg3+arg6,2)
 10           call pplot(arg1-arg4,arg2+arg5,arg3+arg6,2)
 11           call pplot(arg1-arg4,arg2-arg5,arg3+arg6,2)
 12           call pline(arg1+arg4,arg2-arg5,arg3-arg6,arg1+arg4,arg2-arg5,arg3+arg6,2)
 13           call pline(arg1+arg4,arg2+arg5,arg3-arg6,arg1+arg4,arg2+arg5,arg3+arg6,2)
 14           call pline(arg1-arg4,arg2+arg5,arg3-arg6,arg1-arg4,arg2+arg5,arg3+arg6,2)
 15           return
 16           end
```

```
c
c
c       arrow()
c
c       arrow() draws a sort-of arrow from (0,0,0) to (1,0,0)
c
17      subroutine arrow()

18      call pline(0.,0.,0.,1.,0.,0.,2)
19      call pline(1.,0.,0.,.8,.2,0.,2)
20      call pline(1.,0.,0.,.8,0.,.2,2)
21      call pline(1.,0.,0.,.8,-.2,0.,2)
22      call pline(1.,0.,0.,.8,0.,-.2,2)
23      return
24      end

c
c       pyrmd(arg1,arg2,arg3,arg4,arg5,arg6)
c
c       pyrmd() draws a pyramid with the center of it's base at
c       (arg1,arg2,arg3) and half x,y,z widths of arg4,arg5,arg6.
c       The height is the x half width.
c
25      subroutine pyrmd(arg1,arg2,arg3,arg4,arg5,arg6)
26      real*8 arg1,arg2,arg3,arg4,arg5,arg6

27      real*8 height

28      call pline(arg1-arg4,arg2-arg5,arg3-arg6,arg1+arg4,arg2-arg5,arg3-arg6,2)
29      call pplot(arg1+arg4,arg2+arg5,arg3-arg6,2)
30      call pplot(arg1-arg4,arg2+arg5,arg3-arg6,2)
31      call pplot(arg1-arg4,arg2-arg5,arg3-arg6,2)

32      height = arg4 - arg1
33      call pline(arg1-arg4,arg2-arg5,arg3-arg6,arg1,arg2,arg3+height,2)
34      call pline(arg1+arg4,arg2-arg5,arg3-arg6,arg1,arg2,arg3+height,2)
35      call pline(arg1-arg4,arg2+arg5,arg3-arg6,arg1,arg2,arg3+height,2)
36      call pline(arg1+arg4,arg2+arg5,arg3-arg6,arg1,arg2,arg3+height,2)
37      return
38      end
```

```
      c
      c               subroutine mmult4(mp1,mp2,mpr)
      c
      c               subroutine mmult4 multiplies the mp1 4x4 matrix
      c               and multiplies it by the mp2 4x4 matrix.  the result is
      c               placed in the mpr 4x4 matrix.  internal results are placed
      c               in a temporary matrix, then copied over in order that one of
      c               the operands may be used as the destination matrix
      c
 1            subroutine mmult4(mp1,mp2,mpr)
 2            real*8 mp1(4,4),mp2(4,4),mpr(4,4)
 3            real*8 acc
 4            real*8 temp(4,4)
 5            integer i,j,k

 6            do 100 i=1,4
 7                    do 100 j=1,4
 8                            acc = 0.
 9                            do 110 k=1,4
10                                    acc = acc + mp1(i,k)*mp2(k,j)
11     110                    continue
12                            temp(i,j) = acc
13     100    continue
14            do 120 i=1,4
15                    do 120 j=1,4
16                            mpr(i,j) = temp(i,j)
17     120    continue
18            return
19            end


      c
      c               subroutine mmult1(mp1,mp2,mpr)
      c
      c               subroutine mmult1 multiplies the mp1 4 position vector
      c               by the mp2 4x4 matrix.  the result is put in the mpr 4
      c               position vector.  results are calculated into a temporary
      c               vector, then copied over so that the mp1 vector may be used
      c               as the destination of the result
      c
20            subroutine mmult1(mp1,mp2,mpr)
21            real*8 mp1(4),mp2(4,4),mpr(4)
22            real*8 acc
23            real*8 temp(4)
24            integer i,j,k

25            do 100 j=1,4
26                    acc = 0.
27                    do 110 k=1,4
28                            acc = acc + mp1(k)*mp2(k,j)
29     110            continue
30                    temp(j) = acc
31     100    continue
32            do 120 i=1,4
33            mpr(i) = temp(i)
34     120    continue
35            return
36            end
```

```
c
c       subroutine plot(x,y,penc)
c
c               subroutine plot plots a line from the current pen position to
c               the given pen position using the pencode given.  The possible
c               pen codes are:
c                       2: pen down
c                       3: pen up
c                       999: terminate plotting
c
c               the actual interface described here if for the serial port on the
c               iSBC 86/12a board connected to an HP7225A flat bed plotter.  no
c               handshaking is done.
c
1       subroutine plot(x,y,penc)
2       real*8 x,y
3       integer penc
4       common /penpos/xpos,ypos,pcount
5       real*8 xpos,ypos
6       integer*4 pcount

7       pcount = pcount + 1

8       if(penc .eq. 999) then
9               call putout('P')
10              call putout('U')
11              call putout(';')
12              print *,'the number of points plotted is:',pcount
13              goto 200
14      endif
15      if((xpos.eq.x).and.(ypos.eq.y)) then
16              if(penc .eq. 2) then
17                      call putout('P')
18                      call putout('D')
19                      call putout(';')
20                      call putout('P')
21                      call putout('U')
22                      call putout(';')
23              else
24                      goto 200
25              endif

26      else
27              if(penc .eq. 3) then
28                      call putout('P')
29                      call putout('U')
30              else if(penc .eq. 2) then
31                      call putout('P')
32                      call putout('D')
33              else
34                      call putout('P')
35                      call putout('U')
36                      call putout(';')
37                      goto 200
38              endif
```

```
39                      call putout(';')
40                      call putout('P')
41                      call putout('A')
42                      if(x .gt. 12) x = 12
43                      call ponum(x)
44                      call putout(',')
45                      if(y .gt. 10) y = 10
46                      call ponum(y)
47                      call putout(';')
48              endif
49              xpos = x
50              ypos = y
51      200     return
52              end

        c
        c       subroutine ponum(number)
        c
        c               subroutine ponum takes the given double precision real number,
        c               truncates it to integer, then runs the resultant integer out
        c               the iSBC 86/12a serial port.  leading zeros are suppressed.
        c               the maximum number is 99999!!!
        c
53              subroutine ponum(number)
54              real*8 number
55              character lookup(9)
56              logical flag
57              integer multip(5)
58              integer work

59              data lookup/'1','2','3','4','5','6','7','8','9'/
60              data multip/10000,1000,100,10,1/
61              flag = .false.
62              if(number .lt. 0) number = 0.
63              number = number * 800.
64              do 100 i = 1,5
65              work = aint(number / real(multip(i)))
66              if(work .eq. 0) then
67                      if(flag) call putout('0')
68              else
69                      call putout(lookup(work))
70                      flag = .true.
71              endif
72              number = number - work * multip(i)
73      100     continue
74              if(.not. flag) call putout('0')
75              return
76              end
```

```
      c
      c       subroutine plots
      c
      c               subroutine plots initialized the iSBC86/12 board baud rate
      c               generator(really part of the 8253 timer) and serial line.
      c               the given numbers will set it up for 600 baud, 8 bits, no
      c               parity
      c
 77           subroutine plots
 78           common /penpos/xpos,ypos
 79           real*8 xpos,ypos

 80           xpos = 10000.
 81           ypos = 10000.
 82           call output(#0d6h,intl(#0b6h))
 83           call output(#0d4h,intl(#80h))
 84           call output(#0d4h,intl(0))

 85           call output(#0dah,intl(#72h))
 86           call wastet
 87           call output(#0dah,intl(#25h))
 88           call wastet
 89           call output(#0dah,intl(#62h))
 90           call wastet
 91           call output(#0dah,intl(#0ceh))
 92           call wastet
 93           call output(#0dah,intl(#27h))
 94           return
 95           end


      c
      c       subroutine putout(c)
      c
      c               subroutine putout puts the character given out on the iSBC 86/12
      c               board serial line (checks for transmitter empty, loops on not empty,
      c               on empty puts out the character)
      c
 96           subroutine putout(c)
 97           character c
 98           integer*1 status

 99   100             call input(#0dah,status)
100                  status = status .and. 4
101           if(status .eq. 0) goto 100

102           call output(#0d8h,intl(ichar(c)))
103           return
104           end


      c
      c       subroutine wastet
      c
      c               subroutine wastet wastes a little bit of time while the 8253 gets
      c               its act together
      c
105           subroutine wastet
106           return
107           end
```

APPENDIX B

```
run :f5:graph
'define' 1 /
'viewport' 2.5 2.5 2 2 /
'viewpoint' 10 10 10 0 0 0 /
'window' 10 10 /
'cube' 0 0 0 2 2 2 /
'viewport' 7.5 2.5 2 2 /
'rotate' 15 15 15 /
'cube' 0 0 0 2 2 2 /
'viewport' 2.5 7.5 2 2 /
'ident' /
'viewpoint' 10 0 0 0 0 0 /
'cube' 0 0 0 2 2 2 /
'viewport' 7.5 7.5 2 2 /
'rotate' 30 30 30 /
'cube' 0 0 0 2 2 2 /
'enddef' /
'call' 1 /
'end' /
```